Notebook

March 9, 2025

root/final/trimodel_optimised.ipynb

[1]:
```
import torch
from PIL import Image
from transformers import AutoModelForCausalLM, AutoTokenizer
import gradio as gr
import whisper
import faiss
import json
import numpy as np
from sentence_transformers import SentenceTransformer
from transformers import CLIPModel, CLIPProcessor

#
device = "cuda" if torch.cuda.is_available() else "cpu"

#   GLM-4V
tokenizer = AutoTokenizer.from_pretrained("/root/autodl-tmp/glm-4v-9b",␣
 ↪trust_remote_code=True)
model = AutoModelForCausalLM.from_pretrained(
    "/root/autodl-tmp/glm-4v-9b",
    torch_dtype=torch.bfloat16,
    low_cpu_mem_usage=True,
    trust_remote_code=True
).to(device).eval()

#   Whisper
whisper_model = whisper.load_model("base")

#
text_embedding_model = SentenceTransformer("/root/autodl-tmp/all-MiniLM-L6-v2")

#   CLIP
clip_model = CLIPModel.from_pretrained("/root/autodl-tmp/
 ↪clip-vit-base-patch32").to(device)
clip_processor = CLIPProcessor.from_pretrained("/root/autodl-tmp/
 ↪clip-vit-base-patch32")

#
```

```python
def load_text_retrieval_system(index_file, texts_file):
    """
         FAISS
    """
    index = faiss.read_index(index_file)
    with open(texts_file, "r", encoding="utf-8") as f:
        texts = json.load(f)
    return index, texts

#
def load_multimodal_retrieval_system(index_file, texts_file):
    """
         FAISS
    """
    index = faiss.read_index(index_file)
    with open(texts_file, "r", encoding="utf-8") as f:
        image_texts = json.load(f)
    return index, image_texts

#
def retrieve_texts(query, indices, texts_list, k=3):
    """

    """
    all_retrieved_texts = []
    for index, texts in zip(indices, texts_list):
        query_embedding = text_embedding_model.encode([query],␣
 ↪convert_to_tensor=False)
        distances, indices = index.search(query_embedding.astype("float32"), k)
        retrieved_texts = [texts[i] for i in indices[0]]
        all_retrieved_texts.extend(retrieved_texts)
    return all_retrieved_texts

#
def chunk_text(text, max_length=77):
    """
                max_length
    """
    words = text.split()  #
    chunks = []
    current_chunk = []

    for word in words:
        #          max_length
        if len(" ".join(current_chunk + [word])) <= max_length:
            current_chunk.append(word)
        else:
```

```
            #
            chunks.append(" ".join(current_chunk))
            current_chunk = [word]

    #
    if current_chunk:
        chunks.append(" ".join(current_chunk))

    return chunks

#
def retrieve_multimodal_data(query, index, image_texts, k=3):
    """

    """
    #
    chunks = chunk_text(query)

    #
    chunk_embeddings = []
    for chunk in chunks:
        inputs = clip_processor(text=chunk, return_tensors="pt", padding=True,␣
 ↪truncation=True).to(device)
        with torch.no_grad():
            chunk_embedding = clip_model.get_text_features(**inputs).cpu().
 ↪numpy()
        chunk_embeddings.append(chunk_embedding)

    #
    if chunk_embeddings:
        query_text_embedding = np.mean(chunk_embeddings, axis=0)
    else:
        query_text_embedding = np.zeros((1, 512))  #

    #
    query_embedding = np.concatenate([np.zeros((1, 512)),␣
 ↪query_text_embedding], axis=1)  #
    distances, indices = index.search(query_embedding.astype("float32"), k)
    retrieved_data = [image_texts[i] for i in indices[0]]
    return retrieved_data

#
def transcribe_audio(audio_path):
    """

    """
    if not audio_path:
```

```
        return ""
    try:
        transcription = whisper_model.transcribe(audio_path)
        return transcription["text"]
    except Exception as e:
        return f"    : {str(e)}"

#      RAG
def generate_description(image, query, text_indices, text_texts_list,␣
 ↪image_index, image_texts):
    """
         RAG
    """
    if not query.strip():
        return "          "

    #
    retrieved_texts = retrieve_texts(query, text_indices, text_texts_list)
    retrieved_images = retrieve_multimodal_data(query, image_index, image_texts)

    #
    context = "     \n" + "\n".join(retrieved_texts) + "\n\n     \n" + "\n".
 ↪join(
        [item["text"] for item in retrieved_images]
    )
    final_query = f"{context}\n\n   {query}"

    #    +
    if image is not None:
        image = image.convert('RGB')
        inputs = tokenizer.apply_chat_template(
            [{"role": "user", "image": image, "content": final_query}],
            add_generation_prompt=True,
            tokenize=True,
            return_tensors="pt",
            return_dict=True
        ).to(device)
    else:
        inputs = tokenizer.apply_chat_template(
            [{"role": "user", "content": final_query}],
            add_generation_prompt=True,
            tokenize=True,
            return_tensors="pt",
            return_dict=True
        ).to(device)

    #
```

```
    gen_kwargs = {
        "max_new_tokens": 1000,  #       token
        "do_sample": True,
        "top_k": 1
    }
    with torch.no_grad():
        outputs = model.generate(**inputs, **gen_kwargs)
        outputs = outputs[:, inputs['input_ids'].shape[1]:]
        description = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return description

#
def update_query_from_audio(audio):
    """

    """
    return transcribe_audio(audio)

# Gradio
def gradio_interface(image, transcribed_text, query):
    """
    Gradio
    """
    final_query = query.strip() or transcribed_text.strip()
    if not final_query:
        return "Error: Please enter text or voice input problem."

    description = generate_description(image, final_query, text_indices,␣
 ↪text_texts_list, image_index, image_texts)
    return description

#
def main():
    global text_indices, text_texts_list, image_index, image_texts

    #
    text_index_file_1 = "/root/autodl-tmp/text_index.faiss"  #    FAISS
    text_texts_file_1 = "/root/autodl-tmp/text_texts.json"   #
    text_index_file_2 = "/root/autodl-tmp/NHS_text_index.faiss"  #    FAISS
    text_texts_file_2 = "/root/autodl-tmp/NHS_text_texts.json"   #
    image_index_file = "/root/autodl-tmp/image_index.faiss"    #   FAISS
    image_texts_file = "/root/autodl-tmp/image_texts.json"     #

    #
    text_index_1, text_texts_1 = load_text_retrieval_system(text_index_file_1,␣
 ↪text_texts_file_1)
```

```
    text_index_2, text_texts_2 = load_text_retrieval_system(text_index_file_2,␣
↪text_texts_file_2)
    text_indices = [text_index_1, text_index_2]
    text_texts_list = [text_texts_1, text_texts_2]

    #
    image_index, image_texts =␣
↪load_multimodal_retrieval_system(image_index_file, image_texts_file)

    # Gradio
    with gr.Blocks() as interface:
        gr.Markdown("## GLM-4V Voice + Picture + Text Multimodal Description␣
↪Generation (Integrated with RAG)")
        gr.Markdown("Upload a picture, enter a question, or use voice␣
↪description to let AI generate the corresponding description.")

        with gr.Row():
            image_input = gr.Image(label="Upload a picture (optional)",␣
↪type="pil")

        with gr.Row():
            audio_input = gr.Audio(type="filepath", label="Voice input␣
↪(optional)")
            transcribed_text = gr.Textbox(label="Speech-to-text results␣
↪(editable)", interactive=True)

        with gr.Row():
            query_input = gr.Textbox(label="Input question (can be edited␣
↪manually)", interactive=True)
            submit_button = gr.Button("submit")

        output_text = gr.Textbox(label="Generated Description")

        #
        audio_input.change(update_query_from_audio, inputs=[audio_input],␣
↪outputs=[transcribed_text])

        #         +
        submit_button.click(
            gradio_interface,
            inputs=[image_input, transcribed_text, query_input],
            outputs=[output_text]
        )

    interface.launch(share=True)
```

```
if __name__ == "__main__":
    main()
```

```
Loading checkpoint shards:   0%|          | 0/15 [00:00<?, ?it/s]
```

```
/root/miniconda3/lib/python3.12/site-packages/whisper/__init__.py:150:
FutureWarning: You are using `torch.load` with `weights_only=False` (the current
default value), which uses the default pickle module implicitly. It is possible
to construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  checkpoint = torch.load(fp, map_location=device)
```

```
Running on local URL:  http://127.0.0.1:7860
Running on public URL: https://3e901690f3d64d5b42.gradio.live

This share link expires in 72 hours. For free permanent hosting and GPU
upgrades, run `gradio deploy` from Terminal to deploy to Spaces
(https://huggingface.co/spaces)
```

```
<IPython.core.display.HTML object>
```

This notebook was converted with convert.ploomber.io